RPG description

January 10, 2020

Contents

Note: this documentation is likely to remain incomplete for the foreseeable future.

Note: this project aims more to add features than to maximize test coverage.

1 Introduction

This repository contains the core engine for a multiplayer role-playing game (RPG). Its chief purpose is to simulating the mechanics of the game world, including, eventually, a kind of mechanics of human (and possibly animal) psychology. It does not determine what characters populate the world or what their goals may be. That is up to the players who connect to this engine (see below).

The game is networked and accepts connections by TCP and by websocket. When this aspect of it is of particular interest, the engine will sometimes be referred to as "the server." When on the

other hand the emphasis is on its function as game loop (see below), it will typically be referred to as "the engine."

A Django backend resides in a separate repository. It authenticates users, manages the underlying database, and provides a couple of REST endpoints. The frontend, residing in a third repository, is implemented in Angular. It connects to the Django app via HTTP and to the server via websocket. There is no plan to create a frontend app for the Android or iOS platforms.

2 Game play

2.1 Introduction

Currently the RPG is little more than a chat server that allows characters to move from place to place, and to speak, whisper to, and look at each other. When a character whispers something to another, other characters cannot "hear" the whispered words, but they are notified that the whispering is happening. Similarly for looks.

More than forty bots (§ 3.9) periodically scrape headlines off various websites and "speak" them into the RPG. Although the game also features inanimate things, they cannot yet be moved or taken by characters. Each character has a "home" place associated with it, but this feature is currently unused.

Plans for future game play are discussed in § 4.

The rest of this section consists of an experiment in documentation, one which we would not attempt in the workplace. We want to see what good might come out of using basic set theory and denotational semantics to specify the game formally. Formal specifications are usually given instead in ordinary language plus such modeling tools as UML, *etc.* But the language of mathematics is maximally terse, precise, universal, and independent of whichever programming languages, frameworks, technologies, serialization schemes, *etc.*, may be used to implement the system. Let us see if the effort is worth it.

2.2 Preliminaries

We settle a few points of notation before proceeding. The power set of a set S is denoted $\wp(S)$. The domain and codomain of a function $f : A \to B$ are respectively denoted

$$\begin{aligned} \mathrm{dom}\,(f) &= A \\ \mathrm{cod}\,(f) &= B \end{aligned}$$

An injective function f is denoted $f : A \rightarrowtail B$, a surjective one $f : A \twoheadrightarrow B$.

We freely use currying, uncurrying, and partial application. For example, if $f : A \times B \to C$, then we sometimes use the same letter f to denote the curried function $f : A \to B \to C$, and vice versa.

The null value is denoted by $\perp$, and for any set S we define $S_\perp \equiv S \cup \{\perp\}$. The notation $f : D \to R_\perp$ means that the function f is partial on D. A function $f : A \rightarrowtail B_\perp$ is to be understood as injective only on $f^{-1}(B)$ (unless $\perp \in B$). Similarly, $g : A \twoheadrightarrow B_\perp$ is to be understood as surjective only on $g^{-1}(B)$ (unless $\perp \in B$). Unless noted otherwise, all functions are strict in $\perp$.

The *singleton operator* wraps an element of a set:

$$\varsigma_A : A \rightarrowtail \wp(A)\,, \qquad \varsigma_A\,(x) \equiv \{x\}$$

set name	*meaning*
Dir	Directions in space
PlaceName	Place names
PlaceDesc	Place descriptions
ThingName	Thing names
ThingDesc	Thing descriptions
CharName	Character names
CharDesc	Character descriptions
Speech	Utterances

Table 1: Atomic values

And it is convenient to define the *membership relation* restricted to a set A:

$$\in_A = \{(x,S) \mid \{x\} \subseteq S \text{ and } \{x\} \subseteq A\}$$

2.3 Atomic values

Table 1 shows the various sets of atomic values. They will be used below. Additional details:

- **Desc** is the set of all character strings.
- **Speech** is the set of all character strings.
- **Name** is the set of all character strings each of which:
  - is at least 1 and at most 64 characters long,
  - contains only alphanumeric characters or spaces,
  - contains no space that is not preceded and followed by an alphanumeric character, and
  - is case-insensitive
- **Dir** is defined as:

$$\begin{aligned}\textbf{Dir} \equiv \{\ &\texttt{Up}, && \texttt{Down}, && && \\ &\texttt{North}, && \texttt{South}, && \texttt{East}, && \texttt{West}, \\ &\texttt{Northeast}, && \texttt{Northwest}, && \texttt{Southeast}, && \texttt{Southwest}\ \}\end{aligned}$$

2.4 States

Let Σ be the set of game states. A state $\sigma \in \Sigma$ is completely determined by the *state functions* defined in the following sections.

2.4.1 Address

The spatial setting of the game consists of a collection of *addresses*. Their individual properties are not of interest here, so we introduce ony a set **Address** of abstract objects representing all possible addresses. And for each state $\sigma \in \Sigma$ we define the set

$$\mathbf{Address}_\sigma \subset \mathbf{Address}$$

of addresses that are valid (*i.e.*, present) in σ.

2.4.2 Place

A *place* is the game's smallest geographical unit; it is similar to a chat room. The set of all possible places is denoted **Place**, and for each $\sigma \in \Sigma$ we define the set

$$\mathbf{Place}_\sigma \subset \mathbf{Place}$$

of places valid (*i.e.*, present) in σ. The following families of state functions define the properties of a place.

A place *may* be a part of an address:

$$\mathit{PlaceAddress}_\sigma : \mathbf{Place}_\sigma \twoheadrightarrow \mathbf{Address}_{\sigma\perp}$$

Surjectivity here reflects the rule that each address must have at least one place assigned to it. The name of a place is given by[1]

$$\mathit{PlaceName}_\sigma : \mathbf{Place}_\sigma \rightarrowtail \mathbf{PlaceName}$$

and its (not necessarily unique) description is given by

$$\mathit{PlaceDesc}_\sigma : \mathbf{Place}_\sigma \to \mathbf{PlaceDesc}$$

2.4.3 Things

A *thing* is an inanimate object in the game. The set of all possible things is denoted **Thing**, and for each $\sigma \in \Sigma$ we define the set

$$\mathbf{Thing}_\sigma \subset \mathbf{Thing}$$

of things valid (*i.e.*, present) in σ.

The following families of state functions define the properties of a thing. The name of a thing is given by

$$\mathit{ThingName}_\sigma : \mathbf{Thing}_\sigma \rightarrowtail \mathbf{ThingName}$$

and its (not necessarily unique) description is given by

$$\mathit{ThingDesc}_\sigma : \mathbf{Thing}_\sigma \to \mathbf{ThingDesc}$$

[1] The actual implementation does not require uniqueness of place names, but of the combination of a place's name and its address, if any.

All things must be placed somewhere:

$$\mathit{ThingPlace}_\sigma : \mathbf{Thing}_\sigma \to \mathbf{Place}_\sigma$$

The (inanimate) contents of a given place are given by

$$\mathit{PlaceThings}_\sigma : \mathbf{Place}_\sigma \rightarrowtail \wp(\mathbf{Thing}_\sigma)$$

Consistency requires:

$$\forall \sigma \;.\; \mathit{ThingPlace}_\sigma \circ \mathit{PlaceThings}_\sigma = \varsigma_{\mathbf{Place}_\sigma}$$
$$\forall \sigma \;.\; \mathit{PlaceThings}_\sigma \circ \mathit{ThingPlace}_\sigma = \in_{\mathbf{Thing}_\sigma}$$

2.4.4 Characters

A *character* is a player of the game, either live or automated. The set of all possible characters is denoted **Char**, and for each $\sigma \in \Sigma$ we define the set

$$\mathbf{Char}_\sigma \subset \mathbf{Char}$$

of characters valid (*i.e.*, present) in σ.

The following families of state functions define the properties of a character. The name of a character is given by

$$\mathit{CharName}_\sigma : \mathbf{Char}_\sigma \rightarrowtail \mathbf{CharName}$$

and its (not necessarily unique) description is given by

$$\mathit{CharDesc}_\sigma : \mathbf{Char}_\sigma \to \mathbf{CharDesc}$$

All characters must be placed somewhere:

$$\mathit{CharPlace}_\sigma : \mathbf{Char}_\sigma \to \mathbf{Place}_\sigma$$

The (inanimate) contents of a given place are given by

$$\mathit{PlaceChars}_\sigma : \mathbf{Place}_\sigma \rightarrowtail \wp(\mathbf{Char}_\sigma)$$

Finally, consistency requires:

$$\forall \sigma \;.\; \mathit{CharPlace}_\sigma \circ \mathit{PlaceChars}_\sigma = \varsigma_{\mathbf{Place}_\sigma}$$
$$\forall \sigma \;.\; \mathit{PlaceChars}_\sigma \circ \mathit{CharPlace}_\sigma = \in_{\mathbf{Char}_\sigma}$$

We also define the following helper functions:

- $\mathit{CoPlaceName}_\sigma : \mathbf{Char}_\sigma \to \mathbf{PlaceName}$, defined by

$$\mathit{CoPlaceName}_\sigma = \mathit{PlaceName}_\sigma \circ \mathit{CharPlace}_\sigma$$

- $CoPlaceDesc_\sigma : \mathbf{Char}_\sigma \to \mathbf{PlaceDesc}$, defined by

  $$CoPlaceDesc_\sigma = PlaceDesc_\sigma \circ CharPlace_\sigma$$

- $CoContents_\sigma : \mathbf{Char}_\sigma \to \wp(\mathbf{Thing}_\sigma)$, defined by

  $$CoContents_\sigma = PlaceThings_\sigma \circ CharPlace_\sigma$$

- $CoOccupants_\sigma : \mathbf{Char}_\sigma \to \wp(\mathbf{Char}_\sigma)$, defined by

  $$CoOccupants_\sigma = PlaceChars_\sigma \circ CharPlace_\sigma$$

  where the computed set includes the input argument. That is, $\forall \sigma, c \,.\; c \in CoOccupants_\sigma(c)$.

2.4.5 Exits

An *exit* is a way out of a place into an adjoing place. The set of all possible exits is denoted **Exit**, and for each $\sigma \in \Sigma$ we define the set

$$\mathbf{Exit}_\sigma \subset \mathbf{Exit}$$

of characters valid (*i.e.*, present) in σ.

The following families of state functions define the properties of an exit. The *source* and *destination* of an exit are given respectively by

$$ExitSrc_\sigma : \mathbf{Exit}_\sigma \to \mathbf{Place}_\sigma$$
$$ExitDst_\sigma : \mathbf{Exit}_\sigma \to \mathbf{Place}_\sigma$$

Places are allowed to have no ingress or egress, thus:

$$\forall \sigma \in \Sigma \,.\; ExitSrc_\sigma(\mathbf{Exit}_\sigma) \subseteq \mathbf{Place}_\sigma$$
$$\forall \sigma \in \Sigma \,.\; ExitDst_\sigma(\mathbf{Exit}_\sigma) \subseteq \mathbf{Place}_\sigma$$

The set of exits by which a place may be left is given by

$$PlaceExits_\sigma : \mathbf{Place}_\sigma \rightarrowtail \wp(\mathbf{Exit}_\sigma), \qquad PlaceExits_\sigma \equiv ExitSrc_\sigma^{-1}$$

Exits are edges in the directed multigraph of places, *i.e.*, for every exit there is a place to whose exits it belongs.

$$\forall \sigma \in \Sigma \,.\; PlaceExits_\sigma(\mathbf{Place}_\sigma) = \mathbf{Exit}_\sigma$$

Consistency obviously requires that an exit belong to the exits of its source:[2]

$$\forall \sigma \in \Sigma \,.\; PlaceExits_\sigma \circ ExitSrc_\sigma = \in_{\mathbf{Exit}_\sigma}$$

and that a place be the sole source of its exits:[3]

$$\forall \sigma \in \Sigma \,.\; ExitSrc_\sigma \circ PlaceExits_\sigma = \varsigma_{\mathbf{Place}_\sigma}$$

For convenience we defined the composition:

$$CharExits_\sigma : \mathbf{Char}_\sigma \to \wp(\mathbf{Exit}_\sigma)$$
$$CharExits_\sigma \equiv PlaceExits_\sigma \circ CharPlace_\sigma$$

[2] Spelled out: $\forall e \in \mathbf{Exit}_\sigma \,.\; e \in PlaceExits_\sigma(ExitSrc_\sigma(e))$.

[3] Spelled out: $\forall p \in \mathbf{Place}_\sigma \,.\; ExitSrc_\sigma(PlaceExits_\sigma(eid)) = \{pid\}$.

set name	*set definition*	*example*	*meaning*
Joined	$\{\texttt{Joined}\} \times \mathbf{Char}$	$(\texttt{Joined}, c)$	Character c has joined the game.
Disjoined	$\{\texttt{Disjoined}\} \times \mathbf{Char}$	$(\texttt{Disjoined}, c)$	Character c has quit the game.
Entered	$\{\texttt{Entered}\} \times \mathbf{Char} \times \mathbf{Exit}$	$(\texttt{Entered}, c, e)$	Character c has entered via exit e.
Exited	$\{\texttt{Exited}\} \times \mathbf{Char} \times \mathbf{Exit}$	$(\texttt{Exited}, c, e)$	Character c has exited via exit e.
CharEdited	$\{\texttt{CharEdited}\} \times \mathbf{Char}$	$(\texttt{CharEdited}, c)$	Description of character c has changed.
LookedChar	$\{\texttt{LookedChar}\} \times \mathbf{Char} \times \mathbf{Char}$	$(\texttt{LookedChar}, c_1, c_2)$	Character c_1 looked at character c_2.
LookedThing	$\{\texttt{LookedThing}\} \times \mathbf{Char} \times \mathbf{Thing}$	$(\texttt{LookedThing}, c, t)$	Character c looked at thing t.
Said	$\{\texttt{Said}\} \times \mathbf{Char} \times \mathbf{Speech}$	$(\texttt{Said}, c, s)$	Character c said s.

Table 2: Game events

2.5 Event

An *event* is a significant occurrence in the game. How events are triggered by characters' requests, and which characters get notified of the events, are explained in § 2.8. Table 2 shows the various sets of events. Their union forms the set of all possible events:

$$\begin{aligned}\mathbf{Event} \equiv\ & \mathbf{Joined} \sqcup \mathbf{Disjoined} \sqcup \mathbf{Exited} \sqcup \mathbf{Entered} \sqcup \mathbf{CharEdited} \\ & \sqcup \mathbf{LookedChar} \sqcup \mathbf{LookedThing} \sqcup \mathbf{Said}\end{aligned} \tag{1}$$

2.6 Value Messages

In addition to receiving notifications of events, a character will receive a *value message* for each of its requests for data, as explained in § 2.8. Table 3 shows the various sets of value messages; their union forms the set of all possible value messages:

$$\mathbf{Val} \equiv \mathbf{YouAre} \sqcup \mathbf{Place} \sqcup \mathbf{PlaceThings} \sqcup \mathbf{PlaceChars} \sqcup \mathbf{Thing} \sqcup \mathbf{Char} \tag{2}$$

2.7 Requests

Given a state $\sigma \in \Sigma$, the various requests that a character $c \in \mathbf{Char}_\sigma$ can make are shown in Table 4. Their union forms the set of valid requests for c in σ:

$$\begin{aligned}\mathbf{Req}_{\sigma,c} \equiv\ & \mathbf{WhoAmIReq} \sqcup \mathbf{WhereAmIReq} \sqcup \mathbf{WhatIsHereReq} \sqcup \mathbf{WhoIsHereReq} \\ & \sqcup \mathbf{LookThingReq}_{\sigma,c} \sqcup \mathbf{LookCharReq}_{\sigma,c} \sqcup \mathbf{SayReq} \\ & \sqcup \mathbf{EditMeReq} \sqcup \mathbf{ExitReq}_{\sigma,c} \sqcup \mathbf{QuitReq}\end{aligned}$$

set name	*set definition*	*example*	*meaning*
YouAre	$\{\texttt{YouAre}\} \times \textbf{CharName} \times \textbf{CharDesc}$	$(\texttt{YouAre}, n, d)$	Your name is n and your description is d.
Place	$\{\texttt{Place}\} \times \textbf{PlaceName}$	$(\texttt{Place}, n, d)$	Your location is called n and has description d.
PlaceThings	$\{\texttt{PlaceThings}\} \times \wp(\textbf{Thing} \times \textbf{ThingName})$	$(\texttt{PlaceThings}, \varnothing)$	These are the things at your location.
PlaceChars	$\{\texttt{PlaceChars}\} \times \wp(\textbf{Char} \times \textbf{CharName})$	$(\texttt{PlaceChars}, \varnothing)$	These are the (other) characters at your location.
Thing	$\{\texttt{Thing}\} \times \textbf{Thing} \times \textbf{ThingDesc}$	$(\texttt{Thing}, t, d)$	Thing t has description d.
Char	$\{\texttt{Char}\} \times \textbf{Char} \times \textbf{CharDesc}$	$(\texttt{Char}, c, d)$	Character c has description d.

Table 3: Value messages for characters

set name	*set definition*	*example*	*meaning*
WhoAmIReq	$\{\texttt{WhoAmI}\}$	`WhoAmI`	What is my name and description?
WhereAmIReq	$\{\texttt{WhereAmI}\}$	`WhereAmI`	Where am I?
WhatIsHereReq	$\{\texttt{WhatIsHere}\}$	`WhatIsHere`	What things are in this place?
WhoIsHereReq	$\{\texttt{WhoIsHere}\}$	`WhoIsHere`	Which other characters are in this place?
$\textbf{LookThingReq}_{\sigma,c}$	$\{\texttt{LookThing}\} \times \mathit{CoContents}_{\sigma}(c)$	$(\texttt{LookThing}, t)$	Describe thing t.
$\textbf{LookCharReq}_{\sigma,c}$	$\{\texttt{LookChar}\} \times \mathit{CoOccupants}_{\sigma}(c)$	$(\texttt{LookChar}, c)$	Describe character c.
SayReq	$\{\texttt{Say}\} \times \textbf{Speech}$	$(\texttt{Say}, s)$	Say s out loud.
$\textbf{ExitReq}_{\sigma,c}$	$\{\texttt{Exit}\} \times \mathit{CharExits}_{\sigma}(c)$	$(\texttt{Exit}, e)$	Take exit e.
EditMeReq	$\{\texttt{EditMe}\} \times \textbf{Desc}$	$(\texttt{EditMe}, d)$	Change my description to d.
QuitReq	$\{\texttt{Quit}\}$	`Quit`	Log me out of the game.

Table 4: Requests valid for character c in state σ.

2.8 Dynamics

In a given state $\sigma \in \Sigma$, the effect of a request from character $c \in \mathbf{Char}_\sigma$ might be:

i) to mutate the game's state,

ii) to return a value to character c,

iii) to generate an event, of which certain characters should be notified,

or all of the above. Accordingly, the semantics of play for c in σ are defined by the function[4]

$$\mathscr{P}_\sigma(c) : \mathbf{Req}_{\sigma,c} \to \ \Sigma \times \mathbf{Val}_\perp \times \wp(\mathbf{Event} \times \wp(\mathbf{Char}))$$

The following subsections define the play function $\mathscr{P}_\sigma(c)$ on the various subsets of requests. For brevity, state functions unchanged by a request are elided (*i.e.*, we write down a state function only if a request's action changes it). We also omit quantifiers (*e.g.*, $\forall$).

2.8.1 Who Am I

$$\mathscr{P}_\sigma(c)[\![\texttt{WhoAmI}]\!] = (\sigma, v, \varnothing), \quad \text{where } v = (\texttt{YouAre}, \mathit{CharName}_\sigma(c))$$

2.8.2 Where Am I

$$\mathscr{P}_\sigma(c)[\![\texttt{WhereAmI}]\!] = (\sigma, v, \varnothing), \quad \text{where } v = (\texttt{Place}, \mathit{CoPlaceName}_\sigma(c), \mathit{CoPlaceDesc}_\sigma(c))$$

2.8.3 What Is Here

$$\mathscr{P}_\sigma(c)[\![\texttt{WhatIsHere}]\!] = (\sigma, v, \varnothing)$$

where

$$v = (\texttt{PlaceThings}, \kappa)$$
$$\kappa = \{(t, tn) \mid t \in \mathit{CoContents}_\sigma(c) \ \text{ and } \ tn = \mathit{ThingName}_\sigma(t))\}$$

2.8.4 Who Is Here

$$\mathscr{P}_\sigma(c)[\![\texttt{WhoIsHere}]\!] = (\sigma, v, \varnothing)$$

where

$$v = (\texttt{PlaceChars}, \omega)$$
$$\omega = \{(c', cn') \mid c' \in \mathit{CoOccupants}_\sigma(c) - \{c\} \ \text{ and } \ cn' = \mathit{CharName}_\sigma(c')\}$$

[4] We use semantic double-brackets even though the argument of $\mathscr{P}_\sigma(c)$ is not concrete syntax.

2.8.5 Look at Something

$$\begin{aligned}\mathscr{P}_\sigma(c)[\![(\texttt{LookThing},t)]\!] &= (\sigma, v, \{(\tau,\rho)\}),\\ \text{where } v &= (\texttt{Thing}, t, \mathit{ThingDesc}_\sigma(t))\\ \tau &= (\texttt{LookedThing}, c, t)\\ \rho &= \mathit{CoOccupants}_\sigma(c) - \{c\}\end{aligned}$$

2.8.6 Look at Someone

$$\begin{aligned}\mathscr{P}_\sigma(c)[\![(\texttt{LookChar},c')]\!] &= (\sigma, v, \{(\tau,\rho)\}),\\ \text{where } v &= (\texttt{Char}, c', \mathit{ThingDesc}_\sigma(c'))\\ \tau &= (\texttt{LookedChar}, c, c')\\ \rho &= \mathit{CoOccupants}_\sigma(c) - \{c\}\end{aligned}$$

2.8.7 Take an Exit

$$\begin{aligned}\mathscr{P}_\sigma(c)[\![(\texttt{Exit},e)]\!] &= (\sigma', \bot, \{(\tau_1,\rho_1),(\tau_2,\rho_2)\}),\\ \text{where } \tau_1 &= (\texttt{Entered}, c, e)\\ \rho_1 &= \mathit{CoOccupants}_{\sigma'}(c)\\ \tau_2 &= (\texttt{Exited}, c, e)\\ \rho_2 &= \mathit{CoOccupants}_\sigma(c)\end{aligned}$$

$$\mathit{CharPlace}_{\sigma'}(c') = \begin{cases}\mathit{ExitDst}_\sigma(e), & \text{if } c' = c\\ \mathit{CharPlace}_\sigma(c'), & \text{otherwise}\end{cases}$$

$$\mathit{PlaceChars}_{\sigma'}(p) = \begin{cases}\mathit{PlaceChars}_\sigma(p) - \{c\}, & \text{if } p = \mathit{ExitSrc}_\sigma(e)\\ \mathit{PlaceChars}_\sigma(p) \cup \{c\}, & \text{if } p = \mathit{ExitDst}_\sigma(e)\\ \mathit{PlaceChars}_\sigma(p), & \text{otherwise}\end{cases}$$

2.8.8 Change of description

$$\begin{aligned}\mathscr{P}_\sigma(c)[\![(\texttt{EditMe},d)]\!] &= (\sigma', \bot, \{(\tau,\rho)\})\\ \text{where}\quad \mathit{ThingDesc}_{\sigma'}(c') &= \begin{cases}d, & \text{if } c' = c\\ \mathit{ThingDesc}_\sigma(c'), & \text{otherwise}\end{cases}\\ \tau &= (\texttt{CharEdited}, c)\\ \rho &= \mathit{CoOccupants}_\sigma(c)\end{aligned}$$

set	*example*	*meaning*	*entities*
$\mathbf{AddressId}_\sigma$	$(\texttt{Address}, 42)$	Address number 42	$\mathbf{Address}_\sigma$
$\mathbf{PlaceId}_\sigma$	$(\texttt{Place}, 42)$	Place number 42	$\mathbf{Place}_\sigma$
$\mathbf{ExitId}_\sigma$	$(\texttt{Exit}, 42)$	Exit number 42	$\mathbf{Exit}_\sigma$
$\mathbf{ThingId}_\sigma$	$(\texttt{Thing}, 42)$	Thing number 42	$\mathbf{Thing}_\sigma$
$\mathbf{CharId}_\sigma$	$(\texttt{Char}, 42)$	Character number 42	$\mathbf{Char}_\sigma$

Table 5: Sets of IDs valid in a given state $\sigma \in \Sigma$, and their corresponding entities.

3 Implementation notes

3.1 Identification of entities

To distinguish the game's various entities—places, characters, *etc.*—each is identified by, and is for all practical purposes identical to, a unique ID, which is a pair consisting of a type constant and a natural number. Table 5 shows the sets of IDs. These identifications modify the requests, events, value messages, *etc.*, listed in § 2 in the straightforward way.

3.2 Overview of design

The game engine is implemented in Haskell and imports helper libraries from other repos. The advantages of Haskell include garbage collection, green threads, strict immutability, algebraic data types, type inference, a parametric type system that includes ad hoc polymorphism plus constraints, lazy evaluation and equational reasoning, a surpassingly terse and elegant syntax, libraries rich in convenient abstractions, and, of course, portability. The disadvantages of Haskell include the lags of garbage collection, the profligate use of memory by a lazy language, and an inability to match the speeds of a low-level language such as C, at least for most kinds of computation (though Haskell does do better than the JVM[5] on several measures).

The engine has two overall requirements:

- *Multiplexing.* It must multiplex incoming game requests and linearize them in some order.
- *Consistency.* It must apply them to the state of the game, in the same order. In the case of requests that ask for changes of state, this not only means that updates to the database should happen in the same order, but also that event notifications informing other players of the update also be sent out in the same order.

The engine fulfills these requirements by running a game loop in a dedicated thread and supplying it with:

- A queue of requests, called the **game queue**.
- Exclusive access to the game database;

[5] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/haskell.html`

- *Sending functions* that convey to the desired recipients server messages bearing event notifications or requested data values.

On each iteration, the loop pops a request off the queue, services it, hitting the database if necessary, and uses the appropriate sending functions to notify the appropriate characters of the event that just occurred. The game loop does not, and should not, know how requests make it onto its queue. Nor should it know how sending functions work.

Requests are of course pushed onto the queue by client threads, as discussed below. But here it is important to note that these clients should not be given access either to the database or to the sending functions by which server messages are conveyed to other clients. The reason is that the only way to implement this functionality correctly would be to make the client actions into critical sections, so that calls to the database and sending functions would be atomic. The idiomatic way to achieve this in Haskell would be to somehow embed database calls in the 'STM' monad. But this monad works by attempting retries, so it might easily repeat a particular database call any number of times. Even if the call were idempotent, this would be unacceptably inefficient. For these reasons, the master thread (the one running the game loop) should monopolize access to both the database and the messaging mechanisms (sending functions). In this way can those operations are kept consistent.

We note here in passing that we rule out the use of a distributed stream processor such as Kafka[6], if only because its storage of all events would for our purposes require too much disk space. The possibility of using other languages and runtimes is discussed further in §4.3.

3.3 Drivers

The game engine must of course accept game requests from clients and push them onto the game loop's queue. In principle the engine could allot one socket per playing character. But this design would not scale up well enough. If for example the implementation assigned each socket connection its own thread, then it would lose time context-switching between them. And even if it avoided threads in favor of an event loop (as envisioned in §4.3), the design would still use up too many kernel resources.[7] In addition, since the engine would have to authenticate the new connection before admitting it to the game, the client's character would first have to exist in the database. Thus clients could not spawn new characters.

For these reasons the game engine does not allow individual characters to connect to the game engine directly. Instead it allows itself to be *driven* by other applications, each of which it trusts without authentication, and to each of which it allots one socket connection. These other applications are called **drivers**. A driver is authorized to spawn characters, endows them with goals and behaviors, and multiplexes the messaging between them and the game engine. Thus it acts as a proxy for its many characters, which contribute to game play.

The game needs at least two drivers. One of them accepts TCP connections from clients across the Internet, authenticates them, and of course thereafter proxies the messaging between them and the game engine. This driver, which does not spawn new characters, is called the **live driver**, and

[6] `https://kafka.apache.org/powered-by`

[7] According to `https://stackoverflow.com/questions/8646190/how-much-memory-is-consumed-by-the-linux-kern` 1000 NPCs would need on the order of 0.5 GB of RAM just to hold the kernel structures associated with the TCP connections.

is the way in which live players join the game. As for NPCs, they are spawned by one or more other drivers, which of course do not need to authenticate them.

A driver can augment the game with elements and NPCs of any kind. It might for example supply "sleep-sweepers," NPCs that roam the world and return any sleeping character (derelict player) to its home, so as not to litter the world with unresponsive characters. Or it might supply a taxi service, assuming the notion of cars were already implemented. More interesting possibilities are mentioned in §4.2.

It is important that NPCs interact only via the game engine. If, for example, the game engine allows characters to punch each other, and if a driver decides to make one of its NPCs punch another, the driver must not try to simulate the punch internally. Rather it must send the game engine a request representing the punch, and it cannot conclude that the punch happened until it receives confirmation from the game engine, in the form of a server message.

The connection between driver and engine should carry JSON for now. If and when it becomes a bottleneck, an alternative TCP port can be opened for the same purpose but communicating in raw binary. (But if communication is dominated by long strings of text then this will not bring a big improvement, because in strings there is already a roughly one-to-one correspondence between character and byte.) A driver therefore presents an opportunity for polyglottism. And since actors[8] can be used to model NPCs, a driver might be fittingly implemented in Erlang, Actix, or Akka, though Akka is apparently quite slow[9].

3.4 Driver action

When a driver connects, the engine spawns a thread, called a **driver thread**, in which it runs the **driver action**. This action creates for the driver a queue that is ready to receive server messages, and it prepares a *sending function* that enqueues server messages onto it. It then spawns a companion thread whose job is to pop server messages off this queue and send them back to the driver via the socket. Finally it runs a loop that waits for the driver's requests, preprocesses them as discussed below, and then pushes them onto the game queue.

The requests that a driver can make are as follows, and describe the lifecycle of an NPC:

1. *Create*. Before it can spawn an NPC, a driver first has to ask the game loop to create it, if it does not already exist. This request should include the desired name of the NPC, which should be unique, along with the ID of the place in which to put it, if it does not already exist somewhere else. (The request does not have to include a description of the NPC.) Having received it, the driver action pairs this request with the sending function and pushes the pair onto the game queue. Having received the pair, the game loop creates the character in the database and uses the sending function to return its CID to the driver action's companion thread, which in turn sends it along to the driver.

2. *Join*. Having received the CID, the driver can now ask the engine to make the character join the game.[10] Having received it, the driver action again pairs this request with its sending

[8] `https://en.wikipedia.org/wiki/Actor_model`

[9] `http://www.techempower.com/benchmarks/`

[10] The live driver will not need to create characters, only to request that they join. That is why joining and creation are two separate requests.

function and sends them to the game loop, which in turn activates the character and *associates* its CID with the sending function. The latter then becomes the means by which that character receives the game loop's messages.

3. *Play*. If the previous two steps represent administrative requests by a driver, then this one is non-administrative, in that it represents a play request from a particular character (*e.g.*, look, speak, *etc.*). The driver merely wraps it before passing it along to the game loop.

4. *Player quits*. As part of making the character quit, the game loop dissociates the corresponding CID from the sending function (though it may of course have the same sending function associated elsewhere with other characters managed by this particular driver).

5. *Destroy*. Delete the character with the given CID from the database.

6. *Quit*. The driver decides to disconnect from the game. This scenario is discussed in §3.5.

It is worth emphasizing that although the game engine deals with the concept of drivers, the game loop itself does not, and should not, know about them or their queues; hence the use of sending functions to abstract those details away. As a consequence, a character should be managed by at most one driver, and the responsibility falls on drivers not to attempt to manage a character jointly.

3.5 Driver disconnection

Recall that a live player connects to the engine via the live driver. Ideally, when the player is done playing, he or she sends an explicit quit request, which arrives at the engine via the driver. More realistically, however, a person quits the game abruptly, by, say, closing a browser window or by otherwise killing his or her local client process ungracefully. For this reason, the live driver must take responsibility for detecting even unannounced disconnections and sending the engine quit requests on their behalf. Otherwise the character will remain logged in but inert, and other characters will wonder why it is stuck in one place and not interacting.

Similarly, a driver itself may need to disconnect from the game; perhaps a developer needs to shut it down for an upgrade. In this case the driver too should first send the engine quit requests on behalf of all its characters. This means that a driver should maintain its own internal set of currently managed characters.

But a driver too may disconnect abruptly. It might crash, or its connection to the engine may fail. Since it will not have had a chance to send quit requests on behalf of its many characters, they will all be left in the "dead" state just described, spoiling the game even more.

So, to prevent this undesirable outcome, the engine itself must take responsibility for "cleaning up" after a driver that disconnects ungracefully. This means that the engine too must keep internal sets of CIDs, each associated with its driver. When a driver suddenly disappears, the engine must iterate over the corresponding set and send each character home and make it quit.

The question then is where to put these sets. Since each driver gets its own driver action in the engine, it might make sense to put them inside the actions. An action's companion thread, which watches messages from the game loop, could be tasked with updating the set whenever it sees a message confirming that a character has been allowed to join or leave the game. In case of sudden disconnection, the companion thread would clean up the characters in the set, and would

then expire. The advantage of this scheme is that it keeps details about drivers out of the game loop.

But unfortunately it won't work. Suppose for example that the driver thread receives from the driver a request to make character X join the game, and suppose that the connection dies immediately after the action pushes this request onto the game queue. Then there is nothing to stop the companion thread from cleaning up the characters in its set, and even from expiring altogether, *before* it receives the game loop's confirmation that character X has joined the game. X would then remain permanently logged in, and inert.

Perhaps, then, it is the game loop that should hold the character sets, one per driver. Upon sudden disconnection of its driver, a driver thread would then send the game loop a special request telling it to clean its characters out. The request would have to carry a label identifying the driver uniquely, which means that the game engine would have to have already assigned that label to the driver when the latter first connected. This already shows the disadvantages of the scheme: it complicates types, flows, and the game loop itself.

Fortunately there is a third way: rather than put the character set inside the driver action's companion thread, entrust it to the driver action itself. As soon as the action sees a request to make a character join or quit the game, it updates the set before it pushes the request onto the game queue. In this way the set always represents all the characters of interest to the driver, and if the driver then disconnects abruptly, the cleanup will not miss any of them. Thus the best solution is also the simplest.

3.6 Serialization

Communication is serialized in JSON. The objects defined in § 2 obviously involve redundancy of types. For example, to have defined **YouAre** as $\{\texttt{YouAre}\} \times$ **Name**, eliding the technically unnecessary constant `CharName`, would have produced serializations that are leaner. But we have opted against such premature optimization of payload size, at least for now, because stronger typing helps prevent errors during *de*serialization. For example,

For example, if a character wishes to say "Good morning" to other characters in the same place, it constructs an element of **SayReq**(a subtype of $\mathbf{Req}_{),t}$hat holds the string "Good morning," and sends it to the server. The corresponding client thread pairs the request with this client's unique ID, and pushes the pair onto the game loop's in-queue. The game loop eventually pops the request off and, seeing that it is of type **SayReq**, it queries the database for the set of characters within "earshot" of the requesting client (*i.e.*, those occupying the same place). It then constructs an instance of the **Said** type (a subtype of **Event**). To forward this message to the other characters, it uses their sending functions, which it looks up in its internal dictionary.

3.7 Things, Characters, and Places

Things are instances of type THING, which include a name, a unique ID,[11] and possibly a description. The subtype CHARACTER represents characters, though at this time there is almost no difference between instances of the two. Places are instances of type PLACE, which bears a name,

[11] Unique only among things.

a unique ID,[12] possibly a description, and possibly also an address. As in real life, an address serves as a tag for a set of places that together form a larger location. For example each of the abovementioned scraping bots is currently confined to a specific address, so that it can move only between the places within that address.

Currently the game engine does not distinguish between a live player and a non-player character (NPC). The words "player" and "character" will typically be used to denote live players and NPCs indifferently, but sometimes "player" will mean a live player ; context should make the meaning clear. Similarly, "thing" will be used to mean either an inanimate object, an NPC, or a live player.

3.8 Persistence

Currently the disk-resident database is Postgres, and this is not likely to change, as the relational model suits the game naturally. An earlier version of the engine interposed Redis as a cache, but apparently Postgres does its own caching so well that on average Redis cannot accelerate queries by more than a factor of two.[13] Roughly the same is presumably true of the other well known on-disk databases, *e.g.*, MySQL. §4.3 includes a discussion of ways to make the engine faster.

Currently the game loop waits for a database update to complete just as it waits for a query to return data. There is therefore no point in pooling multiple connections, as the game loop would use only one of them at a time. Another recommendation against pooling is given below.

3.9 Bots

The bots are implemented in Python (a separate repo) and connect to the engine over TCP. See §3.3 for a better way of filling the game with NPCs.

4 Future plans

4.1 Game rules

4.1.1 Nonverbal gestures

The looks that a character can currently give another are neutral. Game mechanics should include a host of others that are pregnant with meaning: smiles, frowns, smirks, glowers, looks of shock, pity, joy, curiosity, confusion, upset, *etc.* As in real life, characters should give each other neutral looks randomly and at some low frequency.

Similarly there should be gestures of the body, and the game should report (as events) whether a character's body language is not neutral or natural, *e.g.*, whether it is stiff with pretense, or slumped with fear, or erect with contempt.

[12] Unique only among places.

[13] See `https://thebuild.com/blog/2015/10/30/dont-assume-postgresql-is-slow/` and `http://code.jjb.cc/benchmarking-postgres-vs-redis`. But see also the conflicting report mentioned below.

4.1.2 Health, physical and psychological

As in many other games, each character should have its own measure of health. But this should be broken down by type, giving a measure of heart rate, blood pressure, rate of breath, *etc.*

More interestingly, each character should have a set of *feelings*: fear, happiness, anger, pride, humiliation, *etc.*, each with its own scale. As in real life, emotions and biological reactions should happen *to* a character, whether he or she wills it or not, so various looks and gestures (§4.1.1) should trigger the consequent psychological effects, positive and negative. That is, they should influence the receiver's feelings. For example, friendly looks and gestures (if they are sincere) should increase a character's psychic well-being. Similarly, the wrong kind of look can have bad effects: they can provoke hysterical fear in some primates, and combat in geese. These dynamics should be part of the game's mechanics regardless of characters' higher-level gameplay(s).

Conversely, if certain emotions become too intense, they should make a character react in certain ways. If enough physical or psychological torment is inflicted on it, then the humiliation should eventually push its level of rage beyond the limit of self-control, and it should act out in some way, perhaps with physical or emotional violence, eg, a slap or a scream. It will be interesting to see the effect of such surprises on live players. On NPCs they should provoke further reactions, positive or negative depending on the particular gameplay. In this way the mechanics of the game should give rise to feedback loops.

These two classes of measure—the physical and the psychological—should affect one another. For example one has heard of people, and animals, dying of grief, and even of joy. And stress is generally thought to raise the likelihood of heart attack. Reactions should be tuned so that they do not lead to chains that "explode" too quickly.

Comments on the implementation of these features are given in §4.3.8.

4.1.3 Gossip

To mimic the less savory aspects of real life, characters should be able to whisper defamatory gossip about one another. And as in real life, unfortunately, once character A hears gossip about character B, A should not try to verify the claims, but rather should change its nonverbal behavior toward B, giving it looks and gestures that reflect the bad-mouthing (§4.1.1). This change in attitude should be permanent, and once a character hears gossip about another, it should spread it to others, depending on its severity.

For implementation details see §4.3.1.

4.1.4 Property and privacy

When a character disconnects from the server, the engine should automatically put it back in its home place while it remains offline.

A permission system should be established, allowing characters to own property and to forbid others from entering. In this way a character's home can be made a refuge from whatever wars, physical or psychological, or going on outside.

4.1.5 Messaging

The game should have a mechanism that allows a character to send a message, such as a letter, to another one. When character A receives a message in the presence of character B (*i.e.*, if they are both in the same place at that time), the game should inform B of it, even if the message does not regard B.

4.1.6 Weapons

Elements of more traditional warfare can be introduced in the form of guns, *etc.* A character would be able to acquire a gun and shoot at a target, including an enemy character. But whether the shot hits is determined by the character's skill. The game could require characters to practice at firing ranges, as a condition for improving the accuracy of their aim. To mimic real life, when it comes time to fire at an actual enemy target later on, a probability, computed as a function of the extent of past practice, would determine whether the shot is successful.

4.1.7 View port

The simulation of firing ranges raises the question of whether the central component of the game app should be a graphical view port, rather than a chat stream, as is currently the case in the Angular frontend. The game is committed to written language as the medium of communication, especially as high-resolution animated graphics, even if quickly rendered, can sometimes amount to bells and whistles that distract from the essence of the game. But here and there it might be nice to have an image to look at, if only one that depicts the current location.

4.1.8 Economics

The game engine can incorporate a monetary system and an economy in which characters trade various goods and services as they try to eke out an existence. Drivers can be added representing various professions; journalism and policing have already been mentioned. Agriculture is another possibility: fields for planting can be added, and a driver can fill them with farmers. The game engine may require characters to eat some kind of food if they are to maintain their health.

And a character can be required to earn a living, using the wages to purchase various necessities, including food and weapons. The ability to do this comfortably can be impaired as part of an enemy's attack on the character, for example by harming the character's reputation.

4.2 Game play

4.2.1 Police, journalists, corruption, and surveillance

A driver can supply the game with NPCs representing the police, or various authorities. Such authorities can become corrupt, making the game quite interesting. They can, for example, victimize a character by planting evidence on it, perhaps to punish it for having blown the whistle on antecedent corruption, or for having practiced (real) journalism.

Similarly, another driver can simulate researchers who publish journal papers which are nonsense but nevertheless peer-reviewed, and which the court system (simulated by yet another driver)

can use, perhaps secretly, to justify nefarious police practices. For example they can target an innocent character for covert defamation or harassment, wait for the target's life to begin falling apart (see the section on Economics below and its discussion of earning a living), wait a little more for the target to react with the predictable rage and despair, use that reaction to label the target as "dangerous," and then use that "finding" to justify the targeting in hindsight, or even to justify an escalation of the torment. Such self-fulfilling dynamics could rely on the nonverbal mechanisms mentioned above, and could also use surveillance to violate the character's privacy.

If a character speaks out publicly about such targeting without being able to prove it, then the state should appoint a mental health professional to falsely certify the victim as a lunatic, arrest it, and confine it forcibly to a prison or a psychiatric asylum.

4.2.2 Virtuous infiltrations

A player (live or NPC) might try to counteract such corruptions by, say, signing up to become a police officer and trying anonymously inform the media of such corruption. Officers of a different kind might try to fight fire with fire, by planting incriminating evidence on corrupt officers, giving them a taste of their own medicine. Whatever mechanisms are introduced into the game, it will be interesting to see if corruption can be cured.

4.2.3 Aggregate measures

Aggregations of individual health levels can be used to assess the well-being of whole populations. If a character is not isolated by ostracism, it should work to keep both the individual and the community in good health during wars, physical or psychological.

4.2.4 Emergence and phase transitions

Political health can be measured by other aggregates, such as the fraction of the population engaged in factionalism, or in vigilantism on behalf of police, or in vigilantism *against* the police, or the fraction that believe the media gullibly, *etc.* Out of a given set of parameters (rate of corruption, rate of gun ownership, rate at which gossip is transmitted, sensitivity of characters to psychological effects) will emerge a certain dynamics, which in time will presumably equilibrate to a certain *phase*, to borrow the language of statistical mechanics. It will be interesting to measure the corresponding relaxation times, and to attempt phase transitions by tweaking the "macroscopic" parameters just mentioned.

4.2.5 Machine learning

A driver can use machine learning to guide the behaviors of its NPCs. For example such algorithms might be used to improve the aim of an NPC as it practices at the firing range.

Another example: the reference above to journalism suggests a driver that supplies a media apparatus. That is, NPCs can be made to consume news reports, be they true or propagandistic. (It is not yet clear whether such a requirement would have to be enforced by the game engine.) Perhaps machine learning, or something more elementary, can be used to shape their reactions to such reports. They may, for example, make them more gullible, ie, more trusting of authorities.

That in turn may make them more inclined to act as vigilantes on behalf of authorities, and to harass one of the innocent victims mentioned above.

4.2.6 Self-fulfilling loopholes

The "legislature" of such a society could pass "laws" (computable rules) permitting a certain class of outrageous acts, such as, for example, putting a child or an elderly person to death if he or she becomes burdensome in some way. This of course would make some individuals angry. Therefore, the legislature could concomitantly pass a law criminalizing anger, or at least anger about such an issue, so that any person who evinces the anger in any way is to be formally classified as dangerous. Having inflicted that classification, the police can then go about harassing and molesting such individuals under the guise of surveillance. This of course sets up a feedback loop that will eventually provoke a certain fraction of the targets to violence, and the outbursts are to be taken as post-hoc proof that the violent individuals really were violent to begin with. Such macabre self-fulfilling prophecies should make the game an interesting challenge, as long as parameters are turned so as not to make the outburst rate too high.

TODO: Cite the Soho Forum debate of August or September 2019, in which the philosophy professor says that any person deemed dangerous should be subjected to "surveillance."

4.3 Implementation changes

4.3.1 Gossip

The SAY request, which the game loop processes by generating a SAID event, should be extended with a subtype request called GOSSIP, from which the game loop generates

4.3.2 Tunability of drivers

Drivers should typically be dynamically tunable. For example, one that fills the game with police agents should allow the game administrator to adjust the density of such agents in the game world, while the game is running. The driver can do this by exposing to the game administrator a small REST API.

4.3.3 The next driver, and the opportunity to use Rust

Among the possible drivers perhaps the lowest-hanging fruit is one that handles incoming TCP connections from live players across the Internet. This would of course remove from the game engine the burden of listening for, accepting, and authenticating those connections. Since the work of managing sockets and serialization is logically "flat," it is well-suited to a low-level language without garbage collection or a heavy runtime. The most compelling at this time is Rust; see below for further discussion. The resulting driver would be small and well focused, concerning itself only with accepting and authenticating TCP connections from the Internet and plugging them into the game engine. The slowdown introduced by this modularization should be insignificant, since live players anyway take time to think about what moves they want to make. But should there be a need to speed it up, the TCP connection between this driver and the game engine could be changed up for a UDS one, as discussed below.

The change will require that REQUESTs within the game engine be somehow associated not just with an individual character, but also with the driver driving that character.

4.3.4 A slowing modularization for the near term

Normally, one seeks to make a game engine faster. But the near-term plan for this one is actually to make it slower—temporarily. When performance is not the controlling criterion, as it currently is not, the first priority of design becomes modularity. And this engine is not yet as modular as it could be, because it interfaces with the database directly, and hence exposes the choice of underlying DB engine (Postgres), busies itself with talking to it, and, as shown above, and may in future burden itself with the intricacies of caching. The code would be cleaner, smaller, and more focused, if this job was instead outsourced to a separate data server (data access layer), which would hide the choice of underlying DB engine, take on the jobs of caching, conversion, and enforcement of any integrity constraints (though one naturally tries to put these in the DB engine itself), and with which the game engine would communicate by REST API, JSON RPC, or whatever is appropriate. By making it wait for the data server, this modular separation will of course slow the game engine down, especially if communication is via REST API. The separation will also introduce the overhead of managing two separate code bases, along with all that that entails: separate compilation, ensuring agreement of schema and message format, the risk that the data server may not stay as "fresh" to the developer as the game engine would, etc.

But the plan also paves the way for interesting gains, now and in future. First there is the question of what language to use for the data server. A glance at benchmarks[14] shows two server frameworks that vastly outperform most others: Rust's Actix, and JVM's Vert.x. The latter brings the possibility of using Kotlin, arguably the nicest JVM language that is statically typed. But unfortunately Kotlin does not teach any interesting concepts: although its concrete syntax is better than Java's and Scala's, at the end of the day it is just another object-oriented language that pays respect to the functional paradigm. (I say this from experience: I have already used Vert.x + Kotlin to implement a small financial web server.)

Rust is much more interesting. Among its many strengths are that it belongs quite fairly to the functional family of languages and yet compiles directly to native code, producing a small and fast executable. It has immutable variables by default, enjoys a thriving ecosystem which includes such packages as Tokio, Hyper, and Actix, offers zero-cost abstractions, beautiful ones in both functional and asynchronous programming. The result is more beautiful than what can the JVM can produce, and typically faster. (For all its speed after JIT-warmup, the JVM heaves and lumbers, while Rust remains nimble and clever.)

But perhaps most importantly, Rust (AFAIK) is the famous inaugurator of linear types, in the form of borrow checking. This is a new paradigm for many developers, and for all its very friendly error messages, the compiler still manages to be a hard taskmaster. And even when it relents, compilation times can be long. So the language takes time to master. But it is worth it. The language forces the programmer to learn sophisticated new skills.

So, the near-term plan is to factor database calls out of the Haskell game engine and into a Rust data server, leaving the game engine to focus on its main task of multiplexing communications. This factoring-out can be done in a phased manner. The data server can start off providing just one service endpoint, say the one that, given the ID of a Thing, provides its name and description. Once this is working, the Haskell engine in production can be easily modified to use it and can continue to use Postgres directly for the other queries and updates. As the Rust engine grows to provide more of the data services, the Haskell engine can continue to be gradually and easily modified

[14] `http://www.techempower.com/benchmarks/`

to use them, even in production. Thus Postgres will eventually be completely factored out of the Haskell game engine. Since the game is currently only text-based and does not need the detailed granularity of, say, a first-person shooter, the communication delays introduced by this factoring out of the data access layer should be outweighed by the gain in modular simplicity.

4.3.5 The potential for high speed in the long term

As the number of players grows, and with it the volume of communication, the performance inefficiencies of this modular design will eventually be felt. There are multiple solutions for speeding things up, and, like the refactoring just described, they can be applied gradually. First, as alluded to above, the data server can use a cache, such as Redis (without of course having to expose that implementation detail to the game engine or any other backend component). Although the online articles mentioned above say that Redis will not improve Postgres' own caching by more than a factor of two, this one from Amazon[15] makes a stronger claim:

> All Redis data resides in the server's main memory, in contrast to databases such as PostgreSQL, Cassandra, MongoDB and others that store most data on disk or on SSDs. In comparison to traditional disk based databases where most operations require a roundtrip to disk, in-memory data stores such as Redis don't suffer the same penalty. They can therefore support an order of magnitude more operations and faster response times. The result is—blazing fast performance with average read or write operations taking less than a millisecond and support for millions of operations per second.

This paragraph should be read carefully. What it calls "an order of magnitude" better is not the latency, but the throughput. The latency it simply calls "faster" (assuming that is what is meant by "response times"). And these claims are conditioned on using Redis not simply to cache some of the data, but to hold "all" of it. For a game with a lot of characters, even if it is only text-based, this could end up requiring more RAM than is available. Still, under such conditions throughput is more important than latency (as long as the latency is not unreasonably bad). As a compromise, Redis can be used not as an in-memory DB but simply as a cache to be used on an as-needed basis. That should bring significant speedups, and again it could be phased in to the Rust server piecemeal, even in production, making the refactoring low-stress.

If this is not fast enough, then Redis can be chucked and the Rust data server can use its own internal cache, in the form of simple dictionaries, as mentioned above. This brings orders of magnitude of speedup over Redis, since it would dispense with the overhead of serialization and networking. (See Peter Norvig's approximate timing table[16] for handy figures.)

But these two overheads would remain between the data server and the Haskell game engine. What if more speed is needed? Then the data server and game engine can be kept on the same machine, and rather than talk to each other over TCP, they can use a Unix Domain Socket. According to this newsgroup posting[17], referenced by this SO posting[18], the Unix Domain Socket reduces latency by two-thirds, and increases throughput by a factor of seven.

[15] `https://aws.amazon.com/redis/`

[16] `http://norvig.com/21-days.html#answers`

[17] `https://lists.freebsd.org/pipermail/freebsd-performance/2005-February/001143.html`

[18] `https://stackoverflow.com/questions/14973942/tcp-loopback-connection-vs-unix-domain-socket-performa`

What if even more speed is needed? Then the Haskell game engine itself can introduce its own internal cache, again in the form of dictionaries internal to the game loop, to be modified by functional updates. Since a cache hit would save a communication to the data server, this should speed the query up by one or more orders of magnitude. Making the dictionaries mutable instead would speed them up even further, but it would also clutter the code up with MVars.

If at this point more speed is needed, then, since Rust is fast and would by this point be using its own internal cache, the bottleneck would presumably be somewhere in the game engine itself. If profiling reveals bottlenecks, then they may be curable by FFI call to custom routines written in C (or Rust). But if profiling identifies no specific bottleneck, then it means we have hit the limits of Haskell itself. It might not be able to context switch between threads fast enough, or its garbage collection may introduce lags, or it may simply not be able to lazily evaluate expressions any faster.

In this case, the functionality of the game engine—chiefly, multiplexing messages—can be pushed into the Rust data server, making the latter a more monolithic game engine of very high speed. Rust's Actix uses the actor model, to which the `ThreadWithMailbox` abstraction maps quite naturally. So the refactoring will not require reinventing the wheel conceptually: Rust will still have a master game loop thread/actor, client connections will still be represented by actors, and so will drivers (see below), both internally and as they are represented within the main game engine.

The challenge will rather be the change's considerable size. Unlike the smooth, piecemeal refactorings envisioned above, this one would have to happen all in one go, since the logic of the game loop—multiplexing multiple clients, including drivers (see below)—would all have to be transferred from Haskell to Rust all together.

It turns out from the discussion of the 'react' function below that the game engine should probably not be implemented in Rust. But if ever those considerations should change, then the reduction in modularity due to such an implementation would not spoil separation of concerns too badly, because a game engine's core function is, after all, logically "flat": all it does is multiplex messages and maybe also talk to the underlying database.

4.3.6 Further comments on caching

Because most messages to clients are limited to carrying names and IDs (which are short) and not descriptions (which can be long), it may be tempting keep the descriptions out of the cache (thus sparing memory) and/or to spare the game loop the work of sending them, by providing them instead via side-channel REST API (exposed by Django, say). Unfortunately this won't work, because gameplay requires that whenever a character looks at another Thing, the other characters in the same place must be immediately notified of the look, which of course is most easily done by the game loop. (Future gameplay may require similar notifications when it is the current Place that is being looked at.) Still, the size of the cache can be managed by purging it of data that is no longer used. For example, if a player is the last to leave a Place, then the cache can be purged of the place's details (name, ID, description, and exits) and of any inanimate Things in it.

The question then is how to maintain consistency with the on-disk DB. One solution is to take the DB connection out of the game loop and to entrust it instead to a separate thread. The game loop would then effect updates to the DB only by sending requests via this cooperating thread. (If possible, make these requests consist simply of the Haskell IO actions representing DB updates.)

For consistency the cooperating thread should access the DB only by a single connection, and

not a pool. Suppose for example that two characters, A and B, are in place 1, and a third, C, is in the adjoining place 2. Character A sends the engine a request to move to place 2, and then it sends another to move back to place 1. The TCP connection from the client will ensure that these requests arrive at the game engine in order, and they will therefore remain in order when the thread corresponding to player A pushes them onto the game queue. The game loop in turn will process them in order. But if it uses a connection pool to talk to the DB, then there is (AFAICT) no guarantee that the corresponding DB commands ("set A's location to 2," and "set A's location to 1"), will arrive at the DB in the correct order. And even if they did, the DB considers them to come from different clients (different ports), so it may execute them in the opposite order. And because these updates are asynchronous, the game loop has meanwhile moved on, having already reported to other players that A moved to 2 and then back to 1. Even if such problems might be avoidable by clever engineering of the client, the server must not count on them. Instead it should enforce the consistency on its own, by using only a single DB connection. (AFAICT, such problems do not arise for a network application that is not required to inform concurrent users of each others' activities, as in a vanilla web app, which is why it can use a pool.)

4.3.7 Digression: an in-memory design

As a matter of interest we can ask whether persistence can be made even faster. Apparently, in addition to making transactions fast, Redis also makes them durable, by keeping a transaction log on disk, writing to it on every update of in-memory data, and making the write operation an append. Since the I/O entailed in appends is sequential, there is no seek delay, making the updates fast. Details can be found here[19]. (A companion article[20] raves about the speeds attainable with Redis. But another online post, which I now cannot find, measured the just-described "embedded" approach as being orders of magnitude faster than Redis.)

The game engine could do the same. That is, after successfully servicing a request, the game loop could immediately write it to the end of a transaction log file on disk. If and when the size of this log file becomes unwieldy, it can be wiped clean, and the in-memory structure representing the current state of the game can be saved to disk instead. From this fresh checkpoint the transaction log, having just been truncated, could then build up again. And the cycle would continue.

But this trick would seem not to work as well if the hard drive were being shared with other processes performing I/O. Moreover it would mean the abandoning of the relational model, at least near the game engine, and hence the loss of a clean data model which could also be used by ancillary apps. So it is hoped that the needs of speed will not require this drastic change. According to Postgres' documentation[21], "An exclusive row-level lock on a specific row is automatically acquired when the row is updated or deleted." This of course increases the throughput of transactions, and we hope it will make them plenty fast.

4.3.8 Psychological dynamics

To implement the involuntary reactions mentioned above, the game engine will have to react to its own 'Event's. One way to arrange for this is to define a function `react :: [Event] -> m`

[19] `https://medium.com/@denisanikin/what-an-in-memory-database-is-and-how-it-persists-data-efficiently-`
[20] `https://medium.com/@denisanikin/asynchronous-processing-with-in-memory-databases-or-how-to-handle-o`
[21] `https://www.postgresql.org/docs/9.1/explicit-locking.html`

`[Request]`, which computes ‘Request‘s for reactions, based on, say, statistics about characters’ histories, and which the game loop could invoke on each ‘Message‘ it has just generated, and whose resulting ‘Request‘ (if any) it would push onto its own in-queue, to be processed shortly thereafter.

For example, say character A has just punched character B. The game loop receives A’s ‘Request‘ for the punch, ‘respond‘ produces the ‘Event‘ indicating that the punch has happened, and the game loop sends this ‘Event‘ to any other players in the vicinity. But before the game loop turns to the next incoming ‘Request‘ on its queue, it calls ‘react‘ on this ”has-punched” ‘Event‘. The ‘react‘ function examines the details of the character’s current biological and psychological state, and decides, deterministically or otherwise, that character B is to suffer an increase in heart rate and should fly into a screaming rage. Accordingly it returns a list of two ‘Request‘s corresponding to these reactions. The game loop then pushes them onto its own in-queue. A few iterations later, these ‘Request‘s are up for processing, and from them the ‘respond‘ function of course generates two ‘Event‘s indicating that the character’s heart rate has increased and that an outburst is happening (the former might be sent only to the character in question, whereas the latter, being a visible outburst, would need to be sent to nearby characters as well). But again, before moving on, the game loop should invoke ‘react‘ on these two ‘Event‘s as well. Supposing that ‘react‘ finds that B’s rage has crossed a certain threshold, it might produce a ‘Request‘ to raise B’s blood pressure. Again the game loop enqueues that ‘Request‘ on its own in-queue. And so the game loop becomes a feedback loop, and ‘react‘ decides whether the feedback is positive or negative, depending on the circumstances.

This is one way of implementing such involuntary reactions. Note that it risks generating reaction ‘Event‘s at too high a rate, perhaps even an exponential one. One way to limit such growth is to ensure that the game loop calls ‘react‘ no more than once per iteration, and to limit ‘react‘ to returning no more than one ‘Request‘, making its signature ‘[Event] -¿ Maybe Request‘.

Another way of implementing the feature is to outsource the ‘react‘ function to its own driver. The game loop would then send input ‘Event‘s to ‘react‘ to this driver via asychronous channel, and the driver would return any resulting ‘Request‘ to the game engine, which would push it onto the master in-queue as it does for any other client. The advantage of this design is that the driver could be written in the kind of higher-level language that is more typically used for such artifical intelligence: eg Lisp, Scheme, Prolog. If it is to react to things said by one character to another, then it might also use one of the Python libraries for natural language processing.

But the approach also has disadvantages. For one thing, the languages just mentioned have implementations that are slower than GHC, and much slower than Rust. So the driver’s pace may not be able to match that of the game engine, and involuntary reactions may become so delayed as to lose correlation with their causes. That is, they would no longer make psychological sense. Delays may also arise because the ‘react‘ function, whatever language it is implemented in, may not be able to make certain decisions without first getting data that can only be gotten from the database, in which case it would either have to query the DB directly or... send an intermediate ‘Request‘ to the game loop.

The only way to match the paces is to make the game loop wait for ‘react‘, ie, to make the call block. This of course would slow down the whole game, especially if the driver used sluggish libraries from the languages just mentioned. The call can be sped up by not using a separate driver and implementing ‘react‘ in the game engine itself, bringing us back to the beginning of this discussion. Since this would mean writing ‘react‘ in the same language as the rest of the game

engine (or in a faster one via FFI), this point effectively becomes a vote against writing the game engine in a fast language such as Rust. As impressive as Rust is, it does not seem suited to the higher-level logics of artificial intelligence. (This Quora post[22] makes the point with more nuance.) But it can still be used to model the more biological reactions of 'react', ie, those that entail just a bit of arithmetic, and it can still be called by Haskell via FFI.

4.3.9 Summary of plans for the game engine

In conclusion, if the foregoing considerations are sound, then:

- The game engine should stay written in Haskell.
- The more biological 'react' functionality can be added to it in Haskell or by Rust FFI.
- The more sophisticated AI-like aspects of 'react' are most easily written in a higher-level language, either Haskell directly in the game engine, or Lisp or Python in a separate driver, if that does not give rise to spoiling delays.
- Rust can still be used for the data server.
- Rust can still be used for any other drivers that might be suited to low-level languages, such as those that run graph algorithms or numerical computations—which indeed could be such parts of AI as machine learning and matrix computations.

4.3.10 Admin app

The drivers' REST APIs, intended for consumption by the game administrator, can form the back-end data sources for an admin app that the administrator could use to survey the running game, tuning its parameters and those of the various drivers, responding to various adminstrative problems, *etc*.This would require the game engine to set up a special dataflow for such administration, sending server messages down it regardless of place or character.

The admin app could implement a map of the world, showing not just places and exits, but also the current locations of things and characters. For DRYness this functionality should be shared with the game app, allowing characters to see the map and use it. For the sake of realism, a character should probably not be allowed to see the locations of other characters, unless of course he or she has the power of abusive surveillance.

The game app is implemented in Angular, so, to be polyglot, this app could be implemented in React.

4.3.11 Auth server

Since this plan envisions the cooperation of many different network applications, it is probably best to have them authenticate a user against a separate, dedicated authentication server that issues JWTs[23], whose claims can also indicate various levels of play privilege within the game. Such a server is already in the works.

TODO: supply a link

[22] `https://www.quora.com/Is-Rust-a-good-language-for-artificial-intelligence`

[23] `https://en.wikipedia.org/wiki/JSON_Web_Token`